Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-frost-04
Published: 29 March 2022
Intended Status: Informational
Expires: 30 September 2022
Authors: D. Connolly
Zcash Foundation
C. Komlo
University of Waterloo, Zcash Foundation
I. Goldberg C. A. Wood
University of Waterloo Cloudflare

Two-Round Threshold Schnorr Signatures with FROST

Abstract

In this draft, we present the two-round signing variant of FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme. FROST signatures can be issued after a threshold number of entities cooperate to issue a signature, allowing for improved distribution of trust and redundancy with respect to a secret key. Further, this draft specifies signatures that are compatible with [RFC8032]. However, unlike [RFC8032], the protocol for producing signatures in this draft is not deterministic, so as to ensure protection against a key-recovery attack that is possible when even only one participant is malicious.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=cfrg.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-frost.

Status of This Memo

at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 30 September 2022.

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft of FROST.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/cfrg/draft-irtf-cfrg-frost. Instructions are on that page as well.

Unlike signatures in a single-party setting, threshold signatures require cooperation among a threshold number of signers each holding a share of a common private key. The security of threshold schemes in general assume that an adversary can corrupt strictly fewer than a threshold number of participants.

This document presents a variant of a Flexible Round-Optimized Schnorr Threshold (FROST) signature scheme originally defined in [FROST20]. FROST reduces network overhead during threshold signing operations while employing a novel technique to protect against forgery attacks applicable to prior Schnorr-based threshold signature constructions. The variant of FROST presented in this document requires two rounds to compute a signature, and implements signing efficiency improvements described by [Schnorr21]. Single-round signing with FROST is out of scope.

For select ciphersuites, the signatures produced by this draft are compatible with [RFC8032]. However, unlike [RFC8032], signatures

produced by FROST are not deterministic, since deriving nonces deterministically allows for a complete key-recovery attack in multi-party discrete logarithm-based signatures, such as FROST.

Key generation for FROST signing is out of scope for this document. However, for completeness, key generation with a trusted dealer is specified in Appendix B.

1.1. Change Log

draft-04

- Added methods to verify VSS commitments and derive group info (#126, #132).
- Changed check for participants to consider only nonnegative numbers (#133).
- Changed sampling for secrets and coefficients to allow the zero element (#130).
- Split test vectors into separate files (#129)
- Update wire structs to remove commitment shares where not necessary (#128)
- Add failure checks (#127)
- Update group info to include each participant's key and clarify how public key material is obtained (#120, #121).
- Define cofactor checks for verification (#118)
- Various editorial improvements and add contributors (#124, #123, #119, #116, #113, #109)

draft-03

- Refactor the second round to use state from the first round (#94).
- Ensure that verification of signature shares from the second round uses commitments from the first round (#94).
- Clarify RFC8032 interoperability based on PureEdDSA (#86).
- Specify signature serialization based on element and scalar serialization (#85).
- Fix hash function domain separation formatting (#83).

- Make trusted dealer key generation deterministic (#104).

- Add additional constraints on participant indexes and nonce usage (#105, #103, #98, #97).

- Apply various editorial improvements.

draft-02

- Fully specify both rounds of FROST, as well as trusted dealer key generation.

- Add ciphersuites and corresponding test vectors, including suites for RFC8032 compatibility.

- Refactor document for editorial clarity.

draft-01

- Specify operations, notation and cryptographic dependencies.

draft-00

- Outline CFRG draft based on draft-komlo-frost.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following notation and terminology are used throughout this document.

- A participant is an entity that is trusted to hold a secret share.

- NUM_SIGNERS denotes the number of participants, and the number of shares that s is split into. This value MUST NOT exceed 2^16-1.

- THRESHOLD_LIMIT denotes the threshold number of participants required to issue a signature. More specifically, at least THRESHOLD_LIMIT shares must be combined to issue a valid signature.

- len(x) is the length of integer input x as an 8-byte, big-endian integer.

*encode_uint16(x): Convert two byte unsigned integer (uint16) x to a 2-byte, big-endian byte string. For example, encode_uint16(310) = [0x01, 0x36].

*|| denotes concatenation, i.e., x || y = xy.

Unless otherwise stated, we assume that secrets are sampled uniformly at random using a cryptographically secure pseudorandom number generator (CSPRNG); see [RFC4086] for additional guidance on the generation of random numbers.

3. Cryptographic Dependencies

FROST signing depends on the following cryptographic constructs:

*Prime-order Group, Section 3.1;

*Cryptographic hash function, Section 3.2;

These are described in the following sections.

3.1. Prime-Order Group

FROST depends on an abelian group G of prime order p. The fundamental group operation is addition + with identity element I. For any elements A and B of the group G, A + B = B + A is also a member of G. Also, for any A in GG, there exists an element -A such that A + (-A) = (-A) + A = I. Scalar multiplication is equivalent to the repeated application of the group operation on an element A with itself r-1 times, this is denoted as r*A = A + ... + A. For any element A, p * A = I. We denote B as the fixed generator of the group. Scalar base multiplication is equivalent to the repeated application of the group operation B with itself r-1 times, this is denoted as ScalarBaseMult(r). The set of scalars corresponds to GF(p), which refer to as the scalar field. This document uses types Element and Scalar to denote elements of the group G and its set of scalars, respectively. We denote equality comparison as == and assignment of values by =.

We now detail a number of member functions that can be invoked on a prime-order group G.

*Order(): Outputs the order of G (i.e. p).

*Identity(): Outputs the identity element of the group (i.e. I).

*RandomScalar(): A member function of G that chooses at random a Scalar element in GF(p).

*RandomNonzeroScalar(): A member function of G that chooses at random a non-zero Scalar element in GF(p).

*SerializeElement(A): A member function of G that maps an Element A to a unique byte array buf of fixed length Ne.

*DeserializeElement(buf): A member function of G that attempts to map a byte array buf to an Element A, and fails if the input is not a valid byte representation of an element of the group. This function can raise a DeserializeError if deserialization fails or A is the identity element of the group; see Section 6 for group-specific input validation steps.

*SerializeScalar(s): A member function of G that maps a Scalar s to a unique byte array buf of fixed length Ns.

*DeserializeScalar(buf): A member function of G that attempts to map a byte array buf to a Scalar s. This function can raise a DeserializeError if deserialization fails; see Section 6 for group-specific input validation steps.

3.2. Cryptographic Hash Function

FROST requires the use of a cryptographically secure hash function, generically written as H, which functions effectively as a random oracle. For concrete recommendations on hash functions which SHOULD BE used in practice, see Section 6. Using H, we introduce three separate domain-separated hashes, H1, H2, and H3, where H1 and H2 map arbitrary inputs to non-zero Scalar elements of the prime-order group scalar field, and H3 is an alias for H with domain separation applied. The details of H1, H2, and H3 vary based on ciphersuite. See Section 6 for more details about each.

4. Helper functions

Beyond the core dependencies, the protocol in this document depends on the following helper operations:

*Schnorr signatures, Section 4.1;

*Polynomial operations, Section 4.2;

*Encoding operations, Section 4.3;

*Signature binding Section 4.4, group commitment Section 4.5, and challenge computation Section 4.6

This sections describes these operations in more detail.

4.1. Schnorr Signature Operations

In the single-party setting, a Schnorr signature is generated with the following operation.

```
schnorr_signature_generate(msg, SK):

Inputs:
- msg, message to be signed, an octet string
- SK, private key, a scalar

Outputs: signature (R, z), a pair of scalar values

def schnorr_signature_generate(msg, SK):
  PK = G.ScalarBaseMult(SK)
  k = G.RandomScalar()
  R = G.ScalarBaseMult(k)

  comm_enc = G.SerializeElement(R)
  pk_enc = G.SerializeElement(PK)
  challenge_input = comm_enc || pk_enc || msg
  c = H2(challenge_input)

  z = k + (c * SK)
  return (R, z)
```

The corresponding verification operation is as follows. Here, h is the cofactor for the group being operated over, e.g. h=8 for the case of Curve25519, h=4 for Ed448, and h=1 for groups such as ristretto255 and secp256k1, etc. This final scalar multiplication MUST be performed when h>1.

```
schnorr_signature_verify(msg, sig, PK):

Inputs:
- msg, signed message, an octet string
- sig, a tuple (R, z) output from schnorr_signature_generate or FROST
- PK, public key, a group element

Outputs: 1 if signature is valid, and 0 otherwise

def schnorr_signature_verify(msg, sig = (R, z), PK):
  comm_enc = G.SerializeElement(R)
  pk_enc = G.SerializeElement(PK)
  challenge_input = comm_enc || pk_enc || msg
  c = H2(challenge_input)

  l = G.ScalarBaseMult(z)
  r = R + (c * PK)
  check = (l - r) * h
  return check == G.Identity()
```

4.2. Polynomial Operations

This section describes operations on and associated with polynomials that are used in the main signing protocol. A polynomial of degree t is represented as a sorted list of t coefficients. A point on the polynomial is a tuple (x, y), where y = f(x). For notational convenience, we refer to the x-coordinate and y-coordinate of a point p as p.x and p.y, respectively.

4.2.1. Evaluation of a polynomial

This section describes a method for evaluating a polynomial f at a particular input x, i.e., y = f(x) using Horner's method.

```
polynomial_evaluate(x, coeffs):

Inputs:
- x, input at which to evaluate the polynomial, a scalar
- coeffs, the polynomial coefficients, a list of scalars

Outputs: Scalar result of the polynomial evaluated at input x

def polynomial_evaluate(x, coeffs):
  value = 0
  for (counter, coeff) in coeffs.reverse():
    if counter == coeffs.len() - 1:
      value += coeff // add the constant term
    else:
      value += coeff
      value *= x

  return value
```

4.2.2. Lagrange coefficients

Lagrange coefficients are used in FROST to evaluate a polynomial f at f(0), given a set of t other points, where f is represented as a set of coefficients.

```
derive_lagrange_coefficient(x_i, L):

Inputs:
- x_i, an x-coordinate contained in L, a scalar
- L, the set of x-coordinates, each a scalar

Outputs: L_i, the i-th Lagrange coefficient

Errors:
- "invalid parameters", if any coordinate is less than or equal to 0

def derive_lagrange_coefficient(x_i, L):
  if x_i = 0:
    raise "invalid parameters"
  for x_j in L:
    if x_j = 0:
      raise "invalid parameters"

  numerator = 1
  denominator = 1
  for x_j in L:
    if x_j == x_i: continue
    numerator *= x_j
    denominator *= x_j - x_i

  L_i = numerator / denominator
  return L_i
```

4.2.3. Deriving the constant term of a polynomial

Secret sharing requires "splitting" a secret, which is represented as a constant term of some polynomial f of degree t. Recovering the constant term occurs with a set of t points using polynomial interpolation, defined as follows.

```
Inputs:
- points, a set of `t` points on a polynomial f, each a tuple of two
  scalar values representing the x and y coordinates

Outputs: The constant term of f, i.e., f(0)

def polynomial_interpolation(points):
  L = []
  for point in points:
    L.append(point.x)

  f_zero = F(0)
  for point in points:
    delta = point.y * derive_lagrange_coefficient(point.x, L)
    f_zero = f_zero + delta

  return f_zero
```

4.3. Commitment List Encoding

This section describes the subroutine used for encoding a list of signer commitments into a bytestring that is used in the FROST protocol.

```
Inputs:
- commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_comm
  where each element in the list indicates the signer index i and thei
  two commitment Element values (hiding_nonce_commitment_i, binding_no
  by signer index.

Outputs: A byte string containing the serialized representation of com

def encode_group_commitment_list(commitment_list):
  encoded_group_commitment = nil
  for (index, hiding_nonce_commitment, binding_nonce_commitment) in co
    encoded_commitment = encode_uint16(index) ||
                         G.SerializeElement(hiding_nonce_commitment) |
                         G.SerializeElement(binding_nonce_commitment)
    encoded_group_commitment = encoded_group_commitment || encoded_com
  return encoded_group_commitment
```

4.4. Binding Factor Computation

This section describes the subroutine for computing the binding factor based on the signer commitment list and message to be signed.

```
Inputs:
- encoded_commitment_list, an encoded commitment list (as computed
  by encode_group_commitment_list)
- msg, the message to be signed (sent by the Coordinator).

Outputs: binding_factor, a Scalar representing the binding factor

def compute_binding_factor(encoded_commitment_list, msg):
  msg_hash = H3(msg)
  rho_input = encoded_commitment_list || msg_hash
  binding_factor = H1(rho_input)
  return binding_factor
```

4.5. Group Commitment Computation

This section describes the subroutine for creating the group commitment from a commitment list.

```
Inputs:
- commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_comm
  commitments issued by each signer, where each element in the list in
  two commitment Element values (hiding_nonce_commitment_i, binding_no
  This list MUST be sorted in ascending order by signer index.
- binding_factor, a Scalar

Outputs: An Element representing the group commitment

def compute_group_commitment(commitment_list, binding_factor):
  group_commitment = G.Identity()
  for (_, hiding_nonce_commitment, binding_nonce_commitment) in commit
    group_commitment = group_commitment + (hiding_nonce_commitment + (
  return group_commitment
```

4.6. Signature Challenge Computation

This section describes the subroutine for creating the per-message challenge.

```
Inputs:
- group_commitment, an Element representing the group commitment
- group_public_key, public key corresponding to the signer secret key
- msg, the message to be signed (sent by the Coordinator).

Outputs: a challenge Scalar value

def compute_challenge(group_commitment, group_public_key, msg):
  group_comm_enc = G.SerializeElement(group_commitment)
  group_public_key_enc = G.SerializeElement(group_public_key)
  challenge_input = group_comm_enc || group_public_key_enc || msg
  challenge = H2(challenge_input)
  return challenge
```

5. Two-Round FROST Signing Protocol

We now present the two-round variant of the FROST threshold signature protocol for producing Schnorr signatures. It involves signer participants and a coordinator. Signing participants are entities with signing key shares that participate in the threshold signing protocol. The coordinator is a distinguished signer with the following responsibilities:

1. Determining which signers will participate (at least THRESHOLD_LIMIT in number);

2. Coordinating rounds (receiving and forwarding inputs among participants); and

3. Aggregating signature shares output by each participant, and publishing the resulting signature.

FROST assumes the selection of all participants, including the dealer, signer, and Coordinator are all chosen external to the protocol. Note that it is possible to deploy the protocol without a distinguished Coordinator; see Section 7.3 for more information.

Because key generation is not specified, all signers are assumed to have the (public) group state that we refer to as "group info" below, and their corresponding signing key shares.

In particular, it is assumed that the coordinator and each signing participant P_i knows the following group info:

*Group public key, denoted PK = G.ScalarMultBase(s), corresponding to the group secret key s. PK is an output from the group's key generation protocol, such as trusted_dealer_keygenor a DKG.

* Public keys for each signer, denoted PK_i = G.ScalarMultBase(sk_i), which are similarly outputs from the group's key generation protocol.

And that each participant with identifier i additionally knows the following:

* Participant is signing key share sk_i, which is the i-th secret share of s.

The exact key generation mechanism is out of scope for this specification. In general, key generation is a protocol that outputs (1) a shared, group public key PK owned by each Signer, and (2) individual shares of the signing key owned by each Signer. In general, two possible key generation mechanisms are possible, one that requires a single, trusted dealer, and the other which requires performing a distributed key generation protocol. We highlight key generation mechanism by a trusted dealer in Appendix B, for reference.

This signing variant of FROST requires signers to perform two network rounds: 1) generating and publishing commitments, and 2) signature share generation and publication. The first round serves for each participant to issue a commitment to a nonce. The second round receives commitments for all signers as well as the message, and issues a signature share with respect to that message. The Coordinator performs the coordination of each of these rounds. At the end of the second round, the Coordinator then performs an aggregation step and outputs the final signature. This complete interaction is shown in Figure 1.

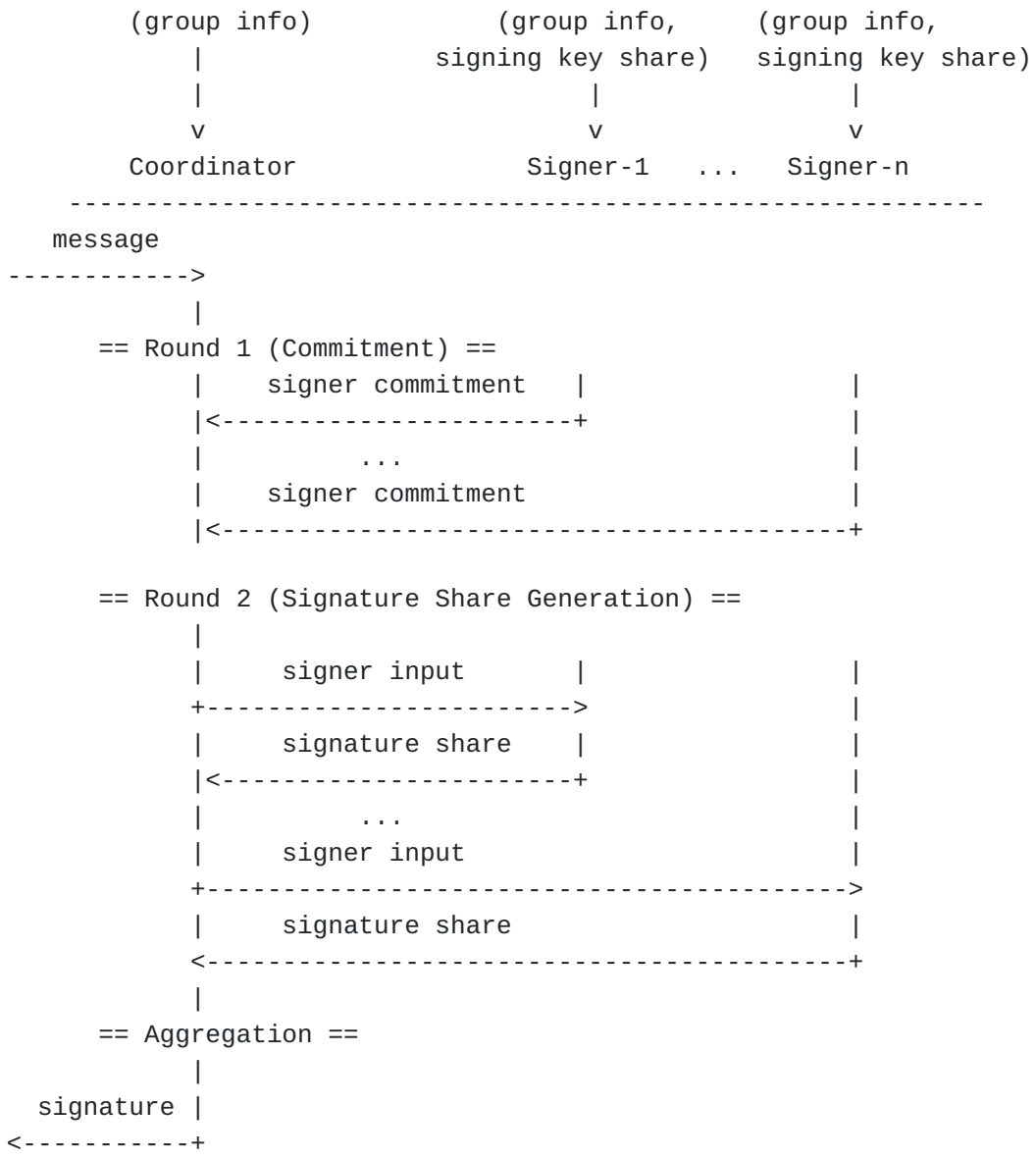

Figure 1: FROST signature overview

Details for round one are described in Section 5.1, and details for round two are described in Section 5.2. The final Aggregation step is described in Section 5.3.

FROST assumes reliable message delivery between Coordinator and signing participants in order for the protocol to complete. Messages exchanged during signing operations are all within the public domain. An attacker masquerading as another participant will result only in an invalid signature; see Section 7.

5.1. Round One - Commitment

Round one involves each signer generating a pair of nonces and their corresponding public commitments. A nonce is a pair of Scalar values, and a commitment is a pair of Element values.

Each signer in round one generates a nonce nonce = (hiding_nonce, binding_nonce) and commitment comm = (hiding_nonce_commitment, binding_nonce_commitment).

Inputs: None

Outputs: (nonce, comm), a tuple of nonce and nonce commitment pairs.

```
def commit():
  hiding_nonce = G.RandomNonzeroScalar()
  binding_nonce = G.RandomNonzeroScalar()
  hiding_nonce_commitment = G.ScalarBaseMult(hiding_nonce)
  binding_nonce_commitment = G.ScalarBaseMult(binding_nonce)
  nonce = (hiding_nonce, binding_nonce)
  comm = (hiding_nonce_commitment, binding_nonce_commitment)
  return (nonce, comm)
```

The private output nonce from Participant P_i is stored locally and kept private for use in the second round. This nonce MUST NOT be reused in more than one invocation of FROST, and it MUST be generated from a source of secure randomness. The public output comm from Participant P_i is sent to the Coordinator; see Appendix C.1 for encoding recommendations.

5.2. Round Two - Signature Share Generation

In round two, the Coordinator is responsible for sending the message to be signed, and for choosing which signers will participate (of number at least THRESHOLD_LIMIT). Signers additionally require locally held data; specifically, their private key and the nonces corresponding to their commitment issued in round one.

The Coordinator begins by sending each signer the message to be signed along with the set of signing commitments for other signers in the participant list. Each signer MUST validate the inputs before processing the Coordinator's request. In particular, the Signer MUST validate commitment_list, deserializing each group Element in the list using DeserializeElement from Section 3.1. If deserialization fails, the Signer MUST abort the protocol. Applications which require that signers not process arbitrary input messages are also required to also perform relevant application-layer input validation checks; see Section 7.4 for more details.

Upon receipt and successful input validation, each Signer then runs the following procedure to produce its own signature share.

```
Inputs:
- index, Index `i` of the signer. Note index will never equal `0`.
- sk_i, Signer secret key share.
- group_public_key, public key corresponding to the signer secret key
- nonce_i, pair of Scalar values (hiding_nonce, binding_nonce) generat
- msg, the message to be signed (sent by the Coordinator).
- commitment_list = [(j, hiding_nonce_commitment_j, binding_nonce_comm
  list of commitments issued in Round 1 by each signer, where each ele
  two commitment Element values (hiding_nonce_commitment_j, binding_no
  This list MUST be sorted in ascending order by signer index.
- participant_list, a set containing identifiers for each signer, simi
  NUM_SIGNERS (sent by the Coordinator).

Outputs: a Scalar value representing the signature share

def sign(index, sk_i, group_public_key, nonce_i, msg, commitment_list,
  # Encode the commitment list
  encoded_commitments = encode_group_commitment_list(commitment_list)

  # Compute the binding factor
  binding_factor = compute_binding_factor(encoded_commitments, msg)

  # Compute the group commitment
  group_commitment = compute_group_commitment(commitment_list, binding

  # Compute Lagrange coefficient
  lambda_i = derive_lagrange_coefficient(index, participant_list)

  # Compute the per-message challenge
  challenge = compute_challenge(group_commitment, group_public_key, ms

  # Compute the signature share
  (hiding_nonce, binding_nonce) = nonce_i
  sig_share = hiding_nonce + (binding_nonce * binding_factor) + (lambd

  return sig_share
```

The output of this procedure is a signature share. Each signer then sends these shares back to the collector; see Appendix C.3 for encoding recommendations. Each signer MUST delete the nonce and corresponding commitment after this round completes.

Upon receipt from each Signer, the Coordinator MUST validate the input signature using DeserializeElement. If validation fails, the Coordinator MUST abort the protocol. If validation succeeds, the Coordinator then verifies the set of signature shares using the following procedure.

5.3. Signature Share Verification and Aggregation

After signers perform round two and send their signature shares to the Coordinator, the Coordinator verifies each signature share for correctness. In particular, for each signer, the Coordinator uses commitment pairs generated during round one and the signature share generated during round two, along with other group parameters, to check that the signature share is valid using the following procedure.

```
Inputs:
- index, Index `i` of the signer. Note index will never equal `0`.
- PK_i, the public key for the ith signer, where `PK_i = G.ScalarBaseM
- comm_i, pair of Element values (hiding_nonce_commitment, binding_non
  in round one from the ith signer.
- sig_share_i, a Scalar value indicating the signature share as produc
- commitment_list = [(j, hiding_nonce_commitment_j, binding_nonce_comm
  issued in Round 1 by each signer, where each element in the list ind
  two commitment Element values (hiding_nonce_commitment_j, binding_no
  This list MUST be sorted in ascending order by signer index.
- participant_list, a set containing identifiers for each signer, simi
  NUM_SIGNERS (sent by the Coordinator).
- group_public_key, the public key for the group
- msg, the message to be signed

Outputs: True if the signature share is valid, and False otherwise.

def verify_signature_share(index, PK_i, comm_i, sig_share_i, commitmen
                           participant_list, group_public_key, msg):
  # Encode the commitment list
  encoded_commitments = encode_group_commitment_list(commitment_list)

  # Compute the binding factor
  binding_factor = compute_binding_factor(encoded_commitments, msg)

  # Compute the group commitment
  group_commitment = compute_group_commitment(commitment_list, binding

  # Compute the commitment share
  (hiding_nonce_commitment, binding_nonce_commitment) = comm_i
  comm_share = hiding_nonce_commitment + (binding_nonce_commitment * b

  # Compute the challenge
  challenge = compute_challenge(group_commitment, group_public_key, ms

  # Compute Lagrange coefficient
  lambda_i = derive_lagrange_coefficient(index, participant_list)

  # Compute relation values
  l = G.ScalarBaseMult(sig_share_i)
  r = comm_share + (PK_i * challenge * lambda_i)

  return l == r

 If any signature share fails to verify, i.e., if
 verify_signature_share returns False for any signer share, the
 Coordinator MUST abort the protocol. Otherwise, if all signer shares
 are valid, the Coordinator performs the aggregate operation and
 publishes the resulting signature.
```

```
Inputs:
- group_commitment, the group commitment returned by compute_group_com
- sig_shares, a set of signature shares z_i for each signer, of length
where THRESHOLD_LIMIT <= NUM_SIGNERS <= MAX_SIGNERS.

Outputs: (R, z), a Schnorr signature consisting of an Element and Scal

def frost_aggregate(group_commitment, sig_shares):
  z = 0
  for z_i in sig_shares:
    z = z + z_i
  return (group_commitment, z)
```

The output signature (R, z) from the aggregation step MUST be encoded as follows:

```
struct {
  opaque R_encoded[Ne];
  opaque z_encoded[Ns];
} Signature;
```

Where Signature.R_encoded is G.SerializeElement(R) and Signature.z_encoded is G.SerializeScalar(z).

6. Ciphersuites

A FROST ciphersuite must specify the underlying prime-order group details and cryptographic hash function. Each ciphersuite is denoted as (Group, Hash), e.g., (ristretto255, SHA-512). This section contains some ciphersuites.

The RECOMMENDED ciphersuite is (ristretto255, SHA-512) Section 6.2. The (Ed25519, SHA-512) ciphersuite is included for backwards compatibility with [RFC8032].

The DeserializeElement and DeserializeScalar functions instantiated for a particular prime-order group corresponding to a ciphersuite MUST adhere to the description in Section 3.1. Validation steps for these functions are described for each the ciphersuites below. Future ciphersuites MUST describe how input validation is done for DeserializeElement and DeserializeScalar.

6.1. FROST(Ed25519, SHA-512)

This ciphersuite uses edwards25519 for the Group and SHA-512 for the Hash function H meant to produce signatures indistinguishable from

Ed25519 as specified in [RFC8032]. The value of the contextString parameter is empty.

*Group: edwards25519 [RFC8032]

-Cofactor (h): 8

-SerializeElement: Implemented as specified in [RFC8032], Section 5.1.2.

-DeserializeElement: Implemented as specified in [RFC8032], Section 5.1.3. Additionally, this function validates that the resulting element is not the group identity element.

-SerializeScalar: Implemented by outputting the little-endian 32-byte encoding of the Scalar value.

-DeserializeScalar: Implemented by attempting to deserialize a Scalar from a 32-byte string. This function can fail if the input does not represent a Scalar between the value 0 and G.Order() - 1.

*Hash (H): SHA-512, and Nh = 64.

-H1(m): Implemented by computing H("rho" || m), interpreting the lower 32 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^252+27742317777372353535851937790883648493.

-H2(m): Implemented by computing H(m), interpreting the lower 32 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^252+27742317777372353535851937790883648493.

-H3(m): Implemented as an alias for H, i.e., H(m).

Normally H2 would also include a domain separator, but for backwards compatibility with [RFC8032], it is omitted.

6.2. FROST(ristretto255, SHA-512)

This ciphersuite uses ristretto255 for the Group and SHA-512 for the Hash function H. The value of the contextString parameter is "FROST-RISTRETTO255-SHA512".

*Group: ristretto255 [RISTRETTO]

-Cofactor (h): 1

- SerializeElement: Implemented using the 'Encode' function from [RISTRETTO].

- DeserializeElement: Implemented using the 'Decode' function from [RISTRETTO].

- SerializeScalar: Implemented by outputting the little-endian 32-byte encoding of the Scalar value.

- DeserializeScalar: Implemented by attempting to deserialize a Scalar from a 32-byte string. This function can fail if the input does not represent a Scalar between the value 0 and G.Order() - 1.

* Hash (H): SHA-512, and Nh = 64.

  - H1(m): Implemented by computing H(contextString || "rho" || m) and mapping the output to a Scalar as described in [RISTRETTO], Section 4.4.

  - H2(m): Implemented by computing H(contextString || "chal" || m) and mapping the output to a Scalar as described in [RISTRETTO], Section 4.4.

  - H3(m): Implemented by computing H(contextString || "digest" || m).

6.3. FROST(Ed448, SHAKE256)

This ciphersuite uses edwards448 for the Group and SHA256 for the Hash function H meant to produce signatures indistinguishable from Ed448 as specified in [RFC8032]. The value of the contextString parameter is empty.

* Group: edwards448 [RFC8032]

  - Cofactor (h): 4

  - SerializeElement: Implemented as specified in [RFC8032], Section 5.2.2.

  - DeserializeElement: Implemented as specified in [RFC8032], Section 5.2.3. Additionally, this function validates that the resulting element is not the group identity element.

  - SerializeScalar: Implemented by outputting the little-endian 48-byte encoding of the Scalar value.

  - DeserializeScalar: Implemented by attempting to deserialize a Scalar from a 48-byte string. This function can fail if the

input does not represent a Scalar between the value 0 and G.Order() - 1.

* Hash (H): SHAKE256, and Nh = 117.

  - H1(m): Implemented by computing H("rho" || m), interpreting the lower 57 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^446 - 13818066809895115352007386748515426880336692474882178609894547503885.

  - H2(m): Implemented by computing H(m), interpreting the lower 57 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^446 - 13818066809895115352007386748515426880336692474882178609894547503885.

  - H3(m): Implemented as an alias for H, i.e., H(m).

Normally H2 would also include a domain separator, but for backwards compatibility with [RFC8032], it is omitted.

6.4. FROST(P-256, SHA-256)

This ciphersuite uses P-256 for the Group and SHA-256 for the Hash function H. The value of the contextString parameter is "FROST-P256-SHA256".

* Group: P-256 (secp256r1) [x9.62]

  - Cofactor (h): 1

  - SerializeElement: Implemented using the compressed Elliptic-Curve-Point-to-Octet-String method according to [SECG].

  - DeserializeElement: Implemented by attempting to deserialize a public key using the compressed Octet-String-to-Elliptic-Curve-Point method according to [SECG], and then performs partial public-key validation as defined in section 5.6.2.3.4 of [KEYAGREEMENT]. This includes checking that the coordinates of the resulting point are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, this function validates that the resulting element is not the group identity element. If these checks fail, deserialization returns an error.

  - SerializeScalar: Implemented using the Field-Element-to-Octet-String conversion according to [SECG].

- DeserializeScalar: Implemented by attempting to deserialize a Scalar from a 32-byte string using Octet-String-to-Field-Element from [SECG]. This function can fail if the input does not represent a Scalar between the value 0 and G.Order() - 1.

* Hash (H): SHA-256, and Nh = 32.

  - H1(m): Implemented using hash_to_field from [HASH-TO-CURVE], Section 5.3 using L = 48, expand_message_xmd with SHA-256, DST = contextString || "rho", and prime modulus equal to Order().

  - H2(m): Implemented using hash_to_field from [HASH-TO-CURVE], Section 5.3 using L = 48, expand_message_xmd with SHA-256, DST = contextString || "chal", and prime modulus equal to Order().

  - H3(m): Implemented by computing H(contextString || "digest" || m).

7. Security Considerations

A security analysis of FROST exists in [FROST20] and [Schnorr21]. The protocol as specified in this document assumes the following threat model.

* Trusted dealer. The dealer that performs key generation is trusted to follow the protocol, although participants still are able to verify the consistency of their shares via a VSS (verifiable secret sharing) step; see Appendix B.2.

* Unforgeability assuming less than (t-1) corrupted signers. So long as an adverary corrupts fewer than (t-1) participants, the scheme remains secure against EUF-CMA attacks.

* Coordinator. We assume the Coordinator at the time of signing does not perform a denial of service attack. A denial of service would include any action which either prevents the protocol from completing or causing the resulting signature to be invalid. Such actions for the latter include sending inconsistent values to signing participants, such as messages or the set of individual commitments. Note that the Coordinator is *not* trusted with any private information and communication at the time of signing can be performed over a public but reliable channel.

The protocol as specified in this document does not target the following goals:

* Post quantum security. FROST, like plain Schnorr signatures, requires the hardness of the Discrete Logarithm Problem.

* Robustness. In the case of failure, FROST requires aborting the protocol.

* Downgrade prevention. The sender and receiver are assumed to agree on what algorithms to use.

* Metadata protection. If protection for metadata is desired, a higher-level communication channel can be used to facilitate key generation and signing.

The rest of this section documents issues particular to implementations or deployments.

7.1. Nonce Reuse Attacks

Nonces generated by each participant in the first round of signing must be sampled uniformly at random and cannot be derived from some deterministic function. This is to avoid replay attacks initiated by other signers, which allows for a complete key-recovery attack. Coordinates MAY further hedge against nonce reuse attacks by tracking signer nonce commitments used for a given group key, at the cost of additional state.

7.2. Protocol Failures

We do not specify what implementations should do when the protocol fails, other than requiring that the protocol abort. Examples of viable failure include when a verification check returns invalid or if the underlying transport failed to deliver the required messages.

7.3. Removing the Coordinator Role

In some settings, it may be desirable to omit the role of the coordinator entirely. Doing so does not change the security implications of FROST, but instead simply requires each participant to communicate with all other participants. We loosely describe how to perform FROST signing among signers without this coordinator role. We assume that every participant receives as input from an external source the message to be signed prior to performing the protocol.

Every participant begins by performing frost_commit() as is done in the setting where a coordinator is used. However, instead of sending the commitment SigningCommitment to the coordinator, every participant instead will publish this commitment to every other participant. Then, in the second round, instead of receiving a SigningPackage from the coordinator, signers will already have sufficient information to perform signing. They will directly perform frost_sign. All participants will then publish a SignatureShare to one another. After having received all signature

shares from all other signers, each signer will then perform frost_verify and then frost_aggregate directly.

The requirements for the underlying network channel remain the same in the setting where all participants play the role of the coordinator, in that all messages that are exchanged are public and so the channel simply must be reliable. However, in the setting that a player attempts to split the view of all other players by sending disjoint values to a subset of players, the signing operation will output an invalid signature. To avoid this denial of service, implementations may wish to define a mechanism where messages are authenticated, so that cheating players can be identified and excluded.

7.4. Input Message Validation

Some applications may require that signers only process messages of a certain structure. For example, in digital currency applications wherein multiple signers may collectively sign a transaction, it is reasonable to require that each signer check the input message to be a syntactically valid transaction. As another example, use of threshold signatures in TLS [TLS] to produce signatures of transcript hashes might require that signers check that the input message is a valid TLS transcript from which the corresponding transcript hash can be derived.

In general, input message validation is an application-specific consideration that varies based on the use case and threat model. However, it is RECOMMENDED that applications take additional precautions and validate inputs so that signers do not operate as signing oracles for arbitrary messages.

8. Contributors

*Isis Lovecruft

*T. Wilson-Brown

*Alden Torres

9. References

9.1. Normative References

[HASH-TO-CURVE] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-14, 18 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-14>.

[KEYAGREEMENT] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar3, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RISTRETTO] Valence, H. D., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-03, 25 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-03>.

[SECG] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <https://secg.org/sec1-v2.pdf>.

[x9.62] ANSI, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62-1998, September 1998.

9.2. Informative References

[FROST20] Komlo, C. and I. Goldberg, "Two-Round Threshold Signatures with FROST", 22 December 2020, <https://eprint.iacr.org/2020/852.pdf>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC

4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/rfc/rfc4086>.

[Schnorr21] Crites, E., Komlo, C., and M. Maller, "How to Prove Schnorr Assuming Schnorr", 11 October 2021, <https://eprint.iacr.org/2021/1375>.

[TLS] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

Appendix A. Acknowledgments

The Zcash Foundation engineering team designed a serialization format for FROST messages which we employ a slightly adapted version here.

Appendix B. Trusted Dealer Key Generation

One possible key generation mechanism is to depend on a trusted dealer, wherein the dealer generates a group secret s uniformly at random and uses Shamir and Verifiable Secret Sharing as described in Sections Appendix B.1 and Appendix B.2 to create secret shares of s to be sent to all other participants. We highlight at a high level how this operation can be performed.

```
Inputs:
- s, a group secret that MUST be derived from at least `Ns` bytes of e
- n, the number of shares to generate, an integer
- t, the threshold of the secret sharing scheme, an integer

Outputs:
- signer_private_keys, `n` shares of the secret key `s`, each a Scalar
- vss_commitment, a vector commitment to each of the coefficients in t

def trusted_dealer_keygen(s, n, t):
  signer_private_keys, coefficients = secret_share_shard(secret_key, n
  vss_commitment = vss_commit(coefficients):
  PK = G.ScalarBaseMult(secret_key)
  return signer_private_keys, vss_commitment
```

It is assumed the dealer then sends one secret key share to each of the NUM_SIGNERS participants, along with C. After receiving their secret key share and C each participant MUST perform vss_verify(secret_key_share_i, C). It is assumed that all participant have the same view of C. The trusted dealer MUST delete the secret_key and secret_key_shares upon completion.

Use of this method for key generation requires a mutually authenticated secure channel between the dealer and participants to

send secret key shares, wherein the channel provides confidentiality and integrity. Mutually authenticated TLS is one possible deployment option.

B.1. Shamir Secret Sharing

In Shamir secret sharing, a dealer distributes a secret s to n participants in such a way that any cooperating subset of t participants can recover the secret. There are two basic steps in this scheme: (1) splitting a secret into multiple shares, and (2) combining shares to reveal the resulting secret.

This secret sharing scheme works over any field F. In this specification, F is the scalar field of the prime-order group G.

The procedure for splitting a secret into shares is as follows.

```
secret_share_shard(s, n, t):

Inputs:
- s, secret to be shared, an element of F
- n, the number of shares to generate, an integer
- t, the threshold of the secret sharing scheme, an integer

Outputs:
- secret_key_shares, A list of n secret shares, which is a tuple
consisting of the participant identifier and the key share, each of
which is an element of F
- coefficients, a vector of the t coefficients which uniquely determin
a polynomial f.

Errors:
- "invalid parameters", if t > n or if t is less than 2

def secret_share_shard(s, n, t):
  if t > n:
    raise "invalid parameters"
  if t < 2:
    raise "invalid parameters"

  # Generate random coefficients for the polynomial, yielding
  # a polynomial of degree (t - 1)
  coefficients = [s]
  for i in range(t - 1):
    coefficients.append(G.RandomScalar())

  # Evaluate the polynomial for each point x=1,...,n
  secret_key_shares = []
  for x_i in range(1, n + 1):
    y_i = polynomial_evaluate(x_i, coefficients)
    secret_key_share_i = (x_i, y_i)
    secret_key_share.append(secret_key_share_i)
  return secret_key_shares, coefficients

 Let points be the output of this function. The i-th element in
 points is the share for the i-th participant, which is the randomly
 generated polynomial evaluated at coordinate i. We denote a secret
 share as the tuple (i, points[i]), and the list of these shares as
 shares. i MUST never equal 0; recall that f(0) = s, where f is the
 polynomial defined in a Shamir secret sharing operation.

 The procedure for combining a shares list of length t to recover the
 secret s is as follows.
```

```
secret_share_combine(shares):

Inputs:
- shares, a list of t secret shares, each a tuple (i, f(i))

Outputs: The resulting secret s, that was previously split into shares

Errors:
- "invalid parameters", if less than t input shares are provided

def secret_share_combine(shares):
  if len(shares) < t:
    raise "invalid parameters"
  s = polynomial_interpolation(shares)
  return s
```

B.2. Verifiable Secret Sharing

Feldman's Verifiable Secret Sharing (VSS) builds upon Shamir secret sharing, adding a verification step to demonstrate the consistency of a participant's share with a public commitment to the polynomial f for which the secret s is the constant term. This check ensure that all participants have a point (their share) on the same polynomial, ensuring that they can later reconstruct the correct secret.

The procedure for committing to a polynomial f of degree t-1 is as follows.

```
vss_commit(coeffs):

Inputs:
- coeffs, a vector of the t coefficients which uniquely determine
a polynomial f.

Outputs: a commitment vss_commitment, which is a vector commitment t
coefficients in coeffs.

def vss_commit(coeffs):
  vss_commitment = []
  for coeff in coeffs:
    A_i = G.ScalarBaseMult(coeff)
    vss_commitment.append(A_i)
  return vss_commitment
```

The procedure for verification of a participant's share is as follows. If vss_verify fails, the participant MUST abort the protocol, and failure should be investigated out of band.

```
vss_verify(share_i, vss_commitment):

Inputs:
- share_i: A tuple of the form (i, sk_i), where i indicates the part
identifier, and sk_i the participant's secret key, where sk_i is a s
the constant term of f.
- vss_commitment: A VSS commitment to a secret polynomial f.

Outputs: 1 if sk_i is valid, and 0 otherwise

vss_verify(share_i, commitment)
  (i, sk_i) = share_i
  S_i = ScalarBaseMult(sk_i)
  S_i' = G.Identity()
  for j in range(0, THRESHOLD_LIMIT-1):
    S_i' += vss_commitment_j * i^j
  if S_i == S_i':
    return 1
  return 0
```

We now define how the coordinator and signing participants can derive group info, which is an input into the FROST signing protocol.

```
derive_group_info(MAX_SIGNERS, THRESHOLD_LIMIT, vss_commitment):

Inputs:
- MAX_SIGNERS, the number of shares to generate, an integer
- THRESHOLD_LIMIT, the threshold of the secret sharing scheme, an in
- vss_commitment: A VSS commitment to a secret polynomial f.

Outputs:
- PK, the public key representing the group
- signer_public_keys, a list of MAX_SIGNERS public keys PK_i for i=1

derive_group_info(MAX_SIGNERS, THRESHOLD_LIMIT, vss_commitment)
  PK = vss_commitment[0]
  signer_public_keys = []
  for i in range(1, MAX_SIGNERS):
    PK_i = G.Identity()
    for j in range(0, THRESHOLD_LIMIT-1):
      PK_i += vss_commitment_j * i^j
    signer_public_keys.append(PK_i)
  return PK, signer_public_keys
```

Appendix C. Wire Format

Applications are responsible for encoding protocol messages between peers. This section contains RECOMMENDED encodings for different protocol messages as described in Section 5.

C.1. Signing Commitment

A commitment from a signer is a pair of Element values. It can be encoded in the following manner.

```
SignerID uint64;

struct {
  SignerID id;
  opaque D[Ne];
  opaque E[Ne];
} SigningCommitment;
```

id The SignerID.

D The commitment hiding factor encoded as a serialized group element.

E The commitment binding factor encoded as a serialized group element.

C.2. Signing Packages

The Coordinator sends "signing packages" to each Signer in Round two. Each package contains the list of signing commitments generated during round one along with the message to sign. This package can be encoded in the following manner.

```
struct {
  SigningCommitment signing_commitments<1..2^16-1>;
  opaque msg<0..2^16-1>;
} SigningPackage;
```

signing_commitments An list of SIGNING_COUNT SigningCommitment values, where THRESHOLD_LIMIT <= SIGNING_COUNT <= NUM_SIGNERS, ordered in ascending order by SigningCommitment.id. This list MUST NOT contain more than one SigningCommitment value corresponding to each signer. Signers MUST ignore SigningPackage values with duplicate SignerIDs.

msg The message to be signed.

C.3. Signature Share

The output of each signer is a signature share which is sent to the Coordinator. This can be constructed as follows.

```
struct {
  SignerID id;
  opaque signature_share[Ns];
} SignatureShare;
```

id The SignerID.

signature_share The signature share from this signer encoded as a serialized scalar.

Appendix D. Test Vectors

This section contains test vectors for all ciphersuites listed in Section 6. All Element and Scalar values are represented in serialized form and encoded in hexadecimal strings. Signatures are represented as the concatenation of their constituent parts. The input message to be signed is also encoded as a hexadecimal string.

Each test vector consists of the following information.

- Configuration: This lists the fixed parameters for the particular instantiation of FROST, including MAX_SIGNERS, THRESHOLD_LIMIT, and NUM_SIGNERS.

- Group input parameters: This lists the group secret key and shared public key, generated by a trusted dealer as described in Appendix B, as well as the input message to be signed. All values are encoded as hexadecimal strings.

- Signer input parameters: This lists the signing key share for each of the NUM_SIGNERS signers.

- Round one parameters and outputs: This lists the NUM_SIGNERS participants engaged in the protocol, identified by their integer index, the hiding and binding commitment values produced in Section 5.1, as well as the resulting group binding factor input, computed in part from the group commitment list encoded as described in Section 4.3, and group binding factor as computed in Section 5.2).

- Round two parameters and outputs: This lists the NUM_SIGNERS participants engaged in the protocol, identified by their integer index, along with their corresponding output signature share as produced in Section 5.2.

- Final output: This lists the aggregate signature as produced in Section 5.3.

D.1. FROST(Ed25519, SHA-512)

```
// Configuration information
MAX_SIGNERS: 3
THRESHOLD_LIMIT: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: 7b1c33d3f5291d85de664833beb1ad469f7fb6025a0ec78b3a7
90c6e13a98304
group_public_key: 15d21ccd7ee42959562fc8aa63224c8851fb3ec85a3faf66040
d380fb9738673
message: 74657374

// Signer input parameters
S1 signer_share: 929dcc590407aae7d388761cddb0c0db6f5627aea8e217f4a033
f2ec83d93509
S2 signer_share: a91e66e012e4364ac9aaa405fcafd370402d9859f7b6685c07ee
d76bf409e80d
S3 signer_share: d3cb090a075eb154e82fdb4b3cb507f110040905468bb9c46da8
bdea643a9a02

// Round one parameters
participants: 1,2
group_binding_factor_input: 000178e175d15cb5cec1257e0d84d797ba8c3dd9b
4c7bc50f3fa527c200bcc6c4a954cdad16ae67ac5919159d655b681bd038574383bab
423614f8967396ee12ca62000288a4e6c3d8353dc3f4aca2e10d10a75fb98d9fbea98
981bfb25375996c5767c932bbf10c41feb17d41cc6433e69f16ccecccc42a00aedf72f
eb5f44929fdf2e2fee26b0dd4af7e749aa1a8ee3c10ae9923f618980772e473f8819a
5d4940e0db27ac185f8a0e1d5f84f88bc887fd67b143732c304cc5fa9ad8e6f57f500
28a8ff
group_binding_factor: c4d7668d793ff4c6ec424fb493cdab3ef5b625eefffe775
71ff28a345e5f700a

// Signer round one outputs
S1 hiding_nonce: 570f27bfd808ade115a701eeee997a488662bca8c2a073143e66
2318f1ed8308
S1 binding_nonce: 6720f0436bd135fe8dddc3fadd6e0d13dbd58a1981e587d377d
48e0b8f1c3c01
S1 hiding_nonce_commitment: 78e175d15cb5cec1257e0d84d797ba8c3dd9b4c7b
c50f3fa527c200bcc6c4a95
S1 binding_nonce_commitment: 4cdad16ae67ac5919159d655b681bd038574383b
ab423614f8967396ee12ca62
S2 hiding_nonce: 2a67c5e85884d0275a7a740ba8f536175271484187973450 71dd
cf1a1bd37206
S2 binding_nonce: a0609158eeb448abe5b0df27f5ece96196df5722c01a999e8a4
5d2d5dfc5620c
S2 hiding_nonce_commitment: 88a4e6c3d8353dc3f4aca2e10d10a75fb98d9fbea
98981bfb25375996c5767c9
S2 binding_nonce_commitment: 32bbf10c41feb17d41cc6433e69f16ccecccc42a0
0aedf72feb5f44929fdf2e2f
```

```
// Round two parameters
participants: 1,2

// Signer round two outputs
S1 sig_share: b7e8f03a1a1149adacb96f952dbc39b6034facceafe4a70d6963592
fce75570c
S2 sig_share: cd388f9aff4376397c5ad231713fe6b167bed9cc88a1cc97b0b6bbe
0316a7909

sig: ebe7efbb42c4b1c55106b5536fb5e9ac7a6d0803ea4ae9c8c629ca51e05c230e
974d8a78fff1ac8e52774a24c00141536b0d869b388674a5191a151000e0d005
```

D.2. FROST(Ed448, SHAKE256)

```
// Configuration information
MAX_SIGNERS: 3
THRESHOLD_LIMIT: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: cdf4a803a21d82fa90692e86541e08d878c9f688e5d71a2bd35
4a9a3af62b8c7c89753055949cab8fd044c17c94211f167672b053659420b00
group_public_key: 800e9b495543b04aaebdba2813de65d1aefe78e8b219d38966b
c0afa1d5d9d685c740c8ab720bff3c84cd9f4a701c1588e40d981f4abb19600
message: 74657374

// Signer input parameters
S1 signer_share: d208a2f1d9ead0cc4b4b9b2e84a22f8e2aa2ab4ee715febe7a08
175d4298dd6bbe2e1c0b29aaa972c78555ea3b3d7308b248994780219e0800
S2 signer_share: d71c9bdf11b81f9f062d08d7b3265744dc7a6014e953e15222bc
8416d5cd0210b4c5e410f90a892c91065fbdae37d51ffc29078acae9f90500
S3 signer_share: dc3094cd49856e71c10e757fe3aa7efa8d5315daea91c4e6c96f
f2cf670328b4a95cad16c96b68e65a87689021323737460b75cc14b2550300

// Round one parameters
participants: 1,2
group_binding_factor_input: 00016d8ef55145bab18c129311f1d07bef2110d0b
6841aae919eb6abf5e523d26f819d3695d78f8aa246c6b6d6fd6c2b8a63dd1cf8e8c8
9a870400a0c29f750605b10c52e347fc538af0d4ebddd23a1e0300482a7d98a39d408
356b9041d5fbaa274c2dc3f248601f21cee912e2f5700c1753a80000242c2fdc11e5f
726d4c897ed118f668a27bfb0d5946b5f513e975638b7c4b0a46cf5184d4a9c1f6310
fd3c10f84d9de704a33aab2af976d60804fa4ecba88458bcf7677a3952f540e20556d
5e90d5aa7e8f226d303ef7b88fb33a63f6cac6a9d638089b1739a5d2564d15fb3e43e
1b0b28a80b54ff7255705a71ee2925e4a3e30e41aed489a579d5595e0df13e32e1e4d
d202a7c7f68b31d6418d9845eb4d757adda6ab189e1bb340db818e5b3bc725d992faf
63e9b0500db10517fe09d3f566fba3a80e46a403e0c7d41548fbf75cf2662b00225b5
02961f98d8c9ff937de0b24c231845
group_binding_factor: 2716e157c3da80b65149b1c2cb546723516272ccf75e111
334533e2840a9bf85f3c71478ade11be26d26d8e4b9a1667af88f7df61670f60a00

// Signer round one outputs
S1 hiding_nonce: 04eccfe12348a5a2e4b30e95efcf4e494ce64b89f6504de46b3d
67a5341baaa931e455c57c6c5c81f4895e333da9d71f7d119fcfbd0d7d2000
S1 binding_nonce: 80bcd1b09e82d7d2ff6dd433b0f81e012cadd4661011c44d929
1269cf24820f5c5086d4363dc67450f24ebe560eb4c2059883545d54aa43a00
S1 hiding_nonce_commitment: 6d8ef55145bab18c129311f1d07bef2110d0b6841
aae919eb6abf5e523d26f819d3695d78f8aa246c6b6d6fd6c2b8a63dd1cf8e8c89a87
0400
S1 binding_nonce_commitment: a0c29f750605b10c52e347fc538af0d4ebddd23a
1e0300482a7d98a39d408356b9041d5fbaa274c2dc3f248601f21cee912e2f5700c17
53a80
S2 hiding_nonce: 3b3bbe82babf2a67ded81b308ba45f73b88f6cf3f6aaa4442256
b7a0354d1567478cfde0a2bba98ba4c3e65645e1b77386eb4063f925e00700
```

```
S2 binding_nonce: bcbd112a88bebf463e3509076c5ef280304cb4f1b3a7499cca1
d5e282cc2010a92ff56a3bdcf5ba352e0f4241ba2e54c1431a895c19fff0600
S2 hiding_nonce_commitment: 42c2fdc11e5f726d4c897ed118f668a27bfb0d594
6b5f513e975638b7c4b0a46cf5184d4a9c1f6310fd3c10f84d9de704a33aab2af976d
6080
S2 binding_nonce_commitment: 4fa4ecba88458bcf7677a3952f540e20556d5e90
d5aa7e8f226d303ef7b88fb33a63f6cac6a9d638089b1739a5d2564d15fb3e43e1b0b
28a80

// Round two parameters
participants: 1,2

// Signer round two outputs
S1 sig_share: c5ab0a80c561d1a616ac70f4f13d993156f65f2b44a4a90f37f0640
7a1b62e3940bf14199301d128358b812bef32cb4bffaf03030238772000
S2 sig_share: 15211cb96d6aa73de803d46caf2043859fd796a6282f9adb00033f1
4f4827f23f8cc792c2e322a1f30631ec7690ac587e5eb9c2afd323e3300

sig: 4d9883057726b029d042418600abe88ad3fec06d6a48dca289482e9d51c10353
37e4d1aae5fd1c73a55701133238602f423886fc134a3c6580e787ce8da00900c1a92
07fd32e9c6f956597202323f8f4264ecfd99e9539ae5c388c8e45c133fb4765ee9ff2
583d90d3e49ba02dff6ab51300
```

D.3. FROST(ristretto255, SHA-512)

```
// Configuration information
MAX_SIGNERS: 3
THRESHOLD_LIMIT: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: b020be204b5e758960458ca9c4675b56b12a8faff2be9c94891
d5e1cd75c880e
group_public_key: e22ac4850672021eac8e0a36dfc4811466fb01108c3427d2347
827467ba02a34
message: 74657374

// Signer input parameters
S1 signer_share: 92ae65bb90030a89507fa00fff08dfed841cf996de5a0c574f1f
4693ddcb6705
S2 signer_share: 611003b3f00bb1e01656ac1818a4419a580e637ecaf67b191521
2e0ae43a470c
S3 signer_share: 439eaa4d36b145e00690c07e5245c5312c00cd65b692ebdbda22
1681eaa92603

// Round one parameters
participants: 1,2
group_binding_factor_input: 0001824e9eddddf02b2a9caf5859825e999d791ca
094f65b814a8bca6013d9cc312774c7e1271d2939a84a9a867e3a06579b4d25659b42
7439ccf0d745b43f75b76600028013834ff4d48e7d6b76c2e732bc611f54720ef8933
c4ca4de7eaaa77ff5cd125e056ecc4f7c4657d3a742354430d768f945db229c335d25
8e9622ad99f3e7582d07b35bd9849ce4af6ad403090d69a7d0eb88bba669a9f985175
d70cd15ad5f1ef5b734c98a32b4aab7b43a57e93fc09281f2e7a207076b31e416ba63
f53d9d
group_binding_factor: f00ae6007f2d74a1507c962cf30006be77596106db28f2d
5443fd66d755e780c

// Signer round one outputs
S1 hiding_nonce: 349b3bb8464a1d87f7d6b56f4559a3f9a6335261a3266089a9b1
2d9d6f6ce209
S1 binding_nonce: ce7406016a854be4291f03e7d24fe30e77994c3465de031515a
4c116f22ca901
S1 hiding_nonce_commitment: 824e9eddddf02b2a9caf5859825e999d791ca094f
65b814a8bca6013d9cc3127
S1 binding_nonce_commitment: 74c7e1271d2939a84a9a867e3a06579b4d25659b
427439ccf0d745b43f75b766
S2 hiding_nonce: 4d66d319f20a728ec3d491cbf260cc6be687bd87cc2b5fdb4d5f
528f65fd650d
S2 binding_nonce: 278b9b1e04632e6af3f1a3c144d07922ffcf5efd3a341b47abc
19c43f48ce306
S2 hiding_nonce_commitment: 8013834ff4d48e7d6b76c2e732bc611f54720ef89
33c4ca4de7eaaa77ff5cd12
S2 binding_nonce_commitment: 5e056ecc4f7c4657d3a742354430d768f945db22
9c335d258e9622ad99f3e758
```

```
// Round two parameters
participants: 1,2

// Signer round two outputs
S1 sig_share: 6a539c3a4ee281879a6fb350d20d53e17473f28cd3409ffc238dafe
8d9330605
S2 sig_share: 1d4e59636ee089bfaf548834b07658216649a37f87f0818d5190aa9
b90957505

sig: 7e92309bf40993141acd5f2c7680a302cc5aa5dd291a833906da8e35bc39b03e
87a1f59dbcc20b474ac43b858284ab02dbbc950c5b31218a751d5a846ac97b0a
```

D.4. FROST(P-256, SHA-256)

```
// Configuration information
MAX_SIGNERS: 3
THRESHOLD_LIMIT: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: 6f090d1393ff53bbcbba036c00b8830ab4546c251dece199eb0
3a6a51a5a5928
group_public_key: 033a2a83f9c9fdfdab7d620f48238a5e6157a8eb1d6c382c7b0
ba95b7c9f69679c
message: 74657374

// Signer input parameters
S1 signer_share: 738552e18ea4f2090597aca6c23c1666845c21c676813f9e2678
6f1e410dcecd
S2 signer_share: 780198af894a90563f7555e183bfa9c25463d767cf159da261ed
379767c14472
S3 signer_share: 7c7dde7d83f02ea37952ff1c45433d1e246b8d0927a9fba69d62
00108e74ba17

// Round one parameters
participants: 1,2
group_binding_factor_input: 000102f34caab210d59324e12ba41f0802d9545f7
f702906930766b86c462bb8ff7f3402b724640ea9e262469f401c9006991ba3247c2c
91b97cdb1f0eeab1a777e24e1e0002037f8a998dfc2e60a7ad63bc987cb27b8abf78a
68bd924ec6adb9f251850cbe711024a4e90422a19dd8463214e997042206c39d3df56
168b458592462090c89dbcf84efca0c54f70a585d6aae28679482b4aed03ae5d38297
b9092ab3376d46fdf55
group_binding_factor: 9df349a9f34bf01627f6b4f8b376e8c8261d55508d1cac2
919cdaf7f9cb20e70

// Signer round one outputs
S1 hiding_nonce: 3da92a503cf7e3f72f62dabedbb3ffcc9f555f1c1e78527940fe
3fed6d45e56f
S1 binding_nonce: ec97c41fc77ae7e795067976b2edd8b679f792abb062e4d0c33
f0f37d2e363eb
S1 hiding_nonce_commitment: 02f34caab210d59324e12ba41f0802d9545f7f702
906930766b86c462bb8ff7f34
S1 binding_nonce_commitment: 02b724640ea9e262469f401c9006991ba3247c2c
91b97cdb1f0eeab1a777e24e1e
S2 hiding_nonce: 06cb4425031e695d1f8ac61320717d63918d3edc7a02fcd3f23a
de47532b1fd9
S2 binding_nonce: 2d965a4ea73115b8065c98c1d95c7085db247168012a834d828
5a7c02f11e3e0
S2 hiding_nonce_commitment: 037f8a998dfc2e60a7ad63bc987cb27b8abf78a68
bd924ec6adb9f251850cbe711
S2 binding_nonce_commitment: 024a4e90422a19dd8463214e997042206c39d3df
56168b458592462090c89dbcf8
```

```
// Round two parameters
participants: 1,2

// Signer round two outputs
S1 sig_share: 0a658fe198caddf5ddc407ad58c4615458f02a58d0c1f7a38e25692
98dc41df0
S2 sig_share: e84d948cfec74b5e7540ad09fd69dcd1570f708f2d8573dbbf08cb0
2bc872c75

sig: 035cfbd148da711bbc823455b682ed01a1be3c5415cf692f4a91b7fe22d1dec3
45f2b3246e979229545304b4b7562e3e25afff9ae7fe476b7f4d2e342c4a4b4a65
```

Authors' Addresses

Deirdre Connolly
Zcash Foundation

Email: durumcrustulum@gmail.com

Chelsea Komlo
University of Waterloo, Zcash Foundation

Email: ckomlo@uwaterloo.ca

Ian Goldberg
University of Waterloo

Email: iang@uwaterloo.ca

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net